VAADIN TRAINING

Layouts

Overview

ComponentContainer

- Component that references other components
- Allows adding, removing and iterating over contained components
- Base type of all Layouts as well as some specialized components such as TabSheet

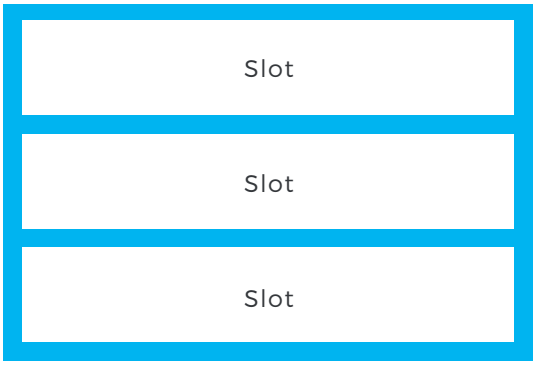
SingleComponentContainer

- Specialized component container that allows defining only one child component
- Base type of **Panel**, **Window** and **UI** classes

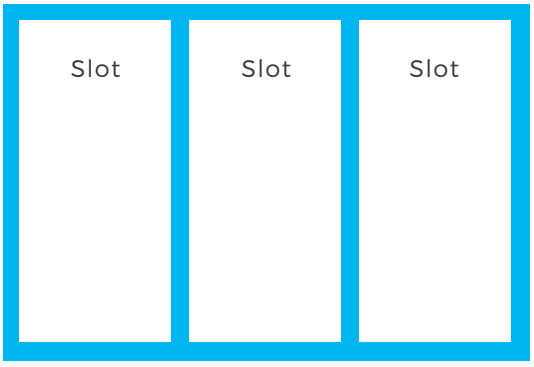
Layout

- Extension of ComponentContainer that defines mechanism for adjusting how child components are aligned and with what kind of spacings and margins
- Layout itself doesn't declare any methods but it has three specializing interfaces: **AlignmentHandler**, **SpacingHandler**, **MarginHandler**
- Selected Layouts implement **ComponentContainer** functionality and various sub types of Layout
- **Vertical**-, **Horizontal**-, and **FormLayout** implement all features of Layout type
- Layout types such as **CssLayout** define less features but allow more flexible customizability via (S)CSS

VERTICAL LAYOUT



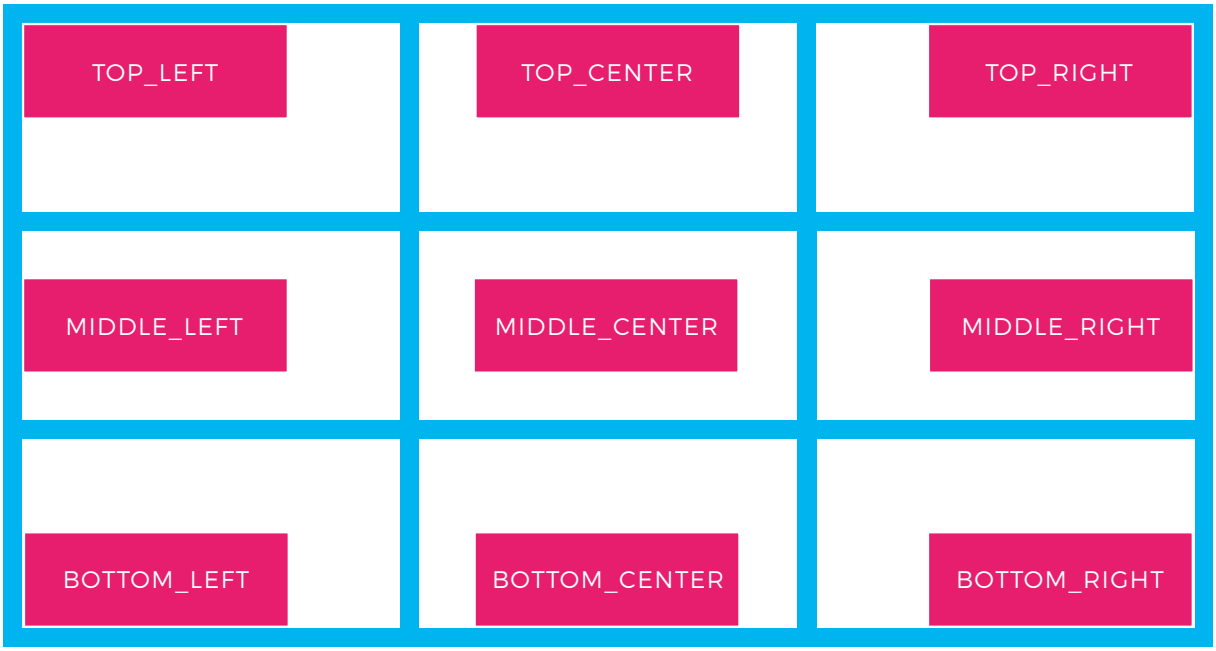
HORIZONTAL LAYOUT



Aligning via AlignmentHandler

- **Vertical**-, and **HorizontalLayout** are the most common basic layouts of Vaadin
- By default these layouts equally divide the space between all child components
- If child component does not use all space available within the layout's slot, the child component can be aligned within the slot with **`setAlignment(Alignment)`** method
- Default alignment can be specified which will be set for all new child components added to this layout

GRID LAYOUT WITH ALIGNED COMPONENTS



ExpandRatio

- **Vertical**-, and **HorizontalLayout** allows defining "expand ratio" that controls how much of available space should be given for each component
- By default the layouts equally distribute the available space for each component
- The space distribution can be altered with expand ratio, often one component is given all available space while others in the same layout take minimum space
- **addComponentsAndExpand** can be used to reduce the amount of configuration code

JAVA

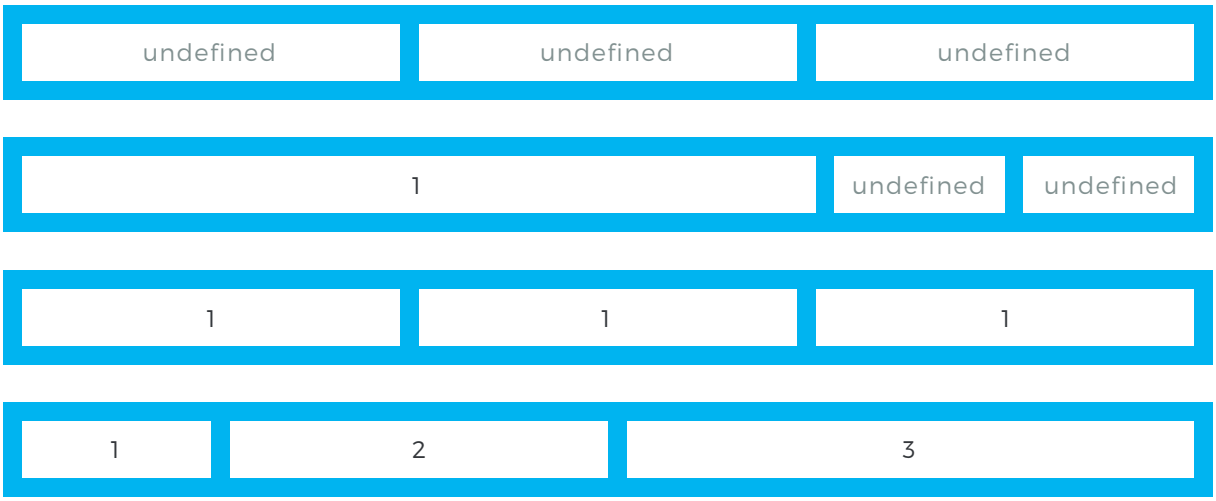
```
addComponentsAndExpand -example:

HorizontalLayout header =
    new HorizontalLayout(title, logout);
VerticalLayout root = new VerticalLayout(header);
root.addComponentsAndExpand(grid);
```

```
// Same result with traditional code:
HorizontalLayout header =
    new HorizontalLayout(title, logout);
VerticalLayout root =
    new VerticalLayout(header, grid);

grid.setSizeFull();
root.setExpandRatio(grid, 1);
root.setSizeFull();
```

EXPAND RATIO EXAMPLES



Navigator

The Navigator is a built-in tool in Vaadin to help you with changing one part of your application out for another when the user wants to navigate. It uses URI fragments, which enable bookmarking views and parameters, and also allows using the browsers back and forward-features.

When you use Vaadin, it is recommended you group your layouts and components into logical groups representing Views. Examples of a View is e.g. `InboxView`, `UserListingView` and `ComposeEmailPopupView.` Each of these are logical wholes, and the user moves from one to another; they have no common components.

The View can consist of the whole visible area of the UI, but more typically excludes the fixed parts such as a menu and a header. As Vaadin can easily replace only a part of the UI, it makes sense not to include the duplicate menus and such with every View, but keep them separate. You 'navigate' from one view to the other by removing and adding the view to a main layout, like this:

JAVA

```
public void navigateTo(View newView){
  mainLayout.removeComponent(currentView);
  mainLayout.addComponent(currentView = newView)

  // optional
  newView.setSizeFull();
  mainLayout.setExpandRatio(newView, 1);
}
```

The Navigator actually has a View interface for this purpose. Each view you want to use with the Navigator should implement it:

JAVA

```
public class DashboardView
  extends CustomComponent
  implements com.vaadin.navigator.View {
    ...
}
```

The interface has only one method, `enter()`. The method is called whenever the Navigator makes that view visible, and is a good place to e.g. load new data from the backend:

JAVA
```
public class MyView extends VerticalLayout implements View {

  @Override
  public void enter(ViewChangeEvent event) {
    // called every time this view is made visible
    refreshTableData();
  }

  ...
}
```

To use the Navigator in your application, you need to do two things in addition to creating the View classes. You need to create a Navigator object for your UI, and you need to register the views you want to use with view names. When creating the Navigator, you decide where in your UI the Views will be placed. The Navigator constructor takes two parameters, where the first one is always the UI, and the second is where you want the Views to be placed:

JAVA
```
public Navigator(UI ui, ComponentContainer container)
public Navigator(UI ui, SingleComponentContainer container)
public Navigator(UI ui, ViewDisplay viewDisplay)

// Example if the second method above:
public class NavigatorUI extends UI {

  @Override
  protected void init(VaadinRequest request) {
    ...
    Navigator navigator = new Navigator(this, this);
    ...
  }
}
```

The example code above would always render each View directly under the UI itself, meaning that the whole UI area is consumed by the View. Most of the time you want to render the View as a sub-section of your UI, because you have menus, headers, etc. visible as well. Thus the second parameter can be any Panel or Layout you want (see Main Layout example later in this document). A `ViewDisplay` alternative is also available, giving you full control over where and how to place views.

When you've created the Navigator, you need to register views. You do this with the `addView()` method in Navigator, that takes a String identifier for the View and a reference to the view itself:

JAVA
```
public class NavigatorUI extends UI {

@Override
protected void init(VaadinRequest request) {
  Navigator navigator = new Navigator(this, this);

  navigator.addView(“dashboard”, new DashboardView());
  // Or this:
  navigator.addView(“dashboard”, DashboardView.class);
  }
}
```

Both methods above bind the `DashboardView` class to the identifier "dashboard", but there is a difference in behaviour. The first method, that receives a View object, keeps that object in memory and re-uses it while navigating. This means any changes you make inside the view will be remembered when the user navigates back to this view.

In the second example, we do not pass an object but the Class of the view. In this case, the Navigator will create a new `DashboardView` object on each navigation, and discarding the old one. This means the View is always rebuilt from scratch. Whichever you should use depends on the view and whether the view should remember settings or not. In either case, the `enter()`-method is called every time the view is made visible.

When you have created the Navigator and registered your Views, you can very easily navigate from one View to the other by calling this method:

JAVA
```
navigator.navigateTo("dashboard");
// or, if you don't have the Navigator reference handy,
UI.getCurrent().getNavigator().navigateTo("dashboard");
```

This tells Navigator to replace the current view with the dashboard and to change the URI fragment to "dashboard" so that the browser knows we've changed 'pages'. The URI looks like this:

`http://localhost:8080/MyApp/#!dashboard`

where:

`#!` : Navigator identifier
`dashboard` : View name

In addition to allowing the user to user the browsers back- and forwards features, the URI fragment can be stored in a bookmark. When creating the Navigator in `UI.init()`, the Navigator will read the fragment and automatically navigate to the correct view.

Navigator also supports adding a parameter String to Views when navigating. You can use this to e.g. automatically load selected data in the `enter()`-method:

JAVA

```
// When we call this
navigator.navigateTo(“dashboard/“ + user.getId()+”/edit”);

// The resulting URI is this:
http://localhost:8080/MyApp/#!dashboard/42/edit

#! : Navigator identifier
dashboard : View name
42/edit : Parameter String

// And we can get the parameter in enter():
public class MyView extends VerticalLayout implements View {
  @Override
  public void enter(ViewChangeEvent event) {
    String params = event.getParameters(); // "42/edit"
    // load user with id 42 from the DB and enable editor
  }
}
```

Our Main Layouts usually uses the Navigator:

JAVA

```
public MainLayout(){
  ...
  Navigator nav = new Navigator(UI.getCurrent(), mainContentWrapper);
  registerViews();
  nav.navigateTo(DashboardView.VIEW_NAME); // String constant to avoid typos
}

public void navigateTo(String viewId){
  nav.navigateTo(viewId);
}
```